COMP 251 - Greedy Algorithms

Algorithms and Data Structures

Written By

David Knapik

McGill University
david.knapik@mcgill.ca

February 26, 2018

Contents

1

Introduction - Scheduling

Greedy algorithms make locally optimal (myopic) choices at each step. They are fast, simple, and easy to code BUT usually are trash.

1.1 Task Scheduling

Suppose a firm receives job orders from n customers. This firm can only do 1 task at a time. Let the time it takes to complete the job of customer i be denoted t_i. Each customer wants their job done as early as possible. Assume the firm processes the jobs in the order $\{1, 2, ..., n\}$ and that the wait time of customer l is

$$w_l = \sum_{i=1}^{l} t_i$$

To maximize the profits, the objective of the firm is to minimize the sum of the waiting times

$$\sum_{l=1}^{n} w_l = \sum_{l=1}^{n} \sum_{i=1}^{l} t_i$$

Algorithm 1.1.1. Greedy Task Scheduling Algorithm:

1. Sort the jobs by length $t_1 \leq t_2 \leq \cdots \leq t_n$
2. Schedule jobs in the order $\{1, 2, ..., n\}$

Theorem 1.1.1. *The above Greedy Task Scheduling Algorithm works and is optimal.*

Proof. A commonly used way to prove that Greedy Algorithms work is by using an exchange argument. We shall do so here.

Let the greedy algorithm schedule in the order $\{1, 2, ..., n\}$. Assume $\exists$ a better schedule $\mathcal{S}$. Then, there is a pair of jobs i, j such that

1. Job i is scheduled immediately before job j by $\mathcal{S}$
2. Job i is longer than job j: $t_i > t_j$

But then, exchanging the order of jobs i, j gives a better schedule $\hat{\mathcal{S}}$. Observe that the waiting time of the other jobs remains the same i.e. $\hat{w_k} = w_k \ \forall k \neq i, j$. But, clearly $\hat{w_i} + \hat{w_j} < w_i + w_j$. This contradicts that $\mathcal{S}$ was optimal. □

1.1.1 Runtime

We just needed to sort n time lengths. Thus, the runtime is $O(n \log n)$. So, the Greedy Task Scheduling algorithm is efficient.

1.2 Class Scheduling (Interval Selection Problem)

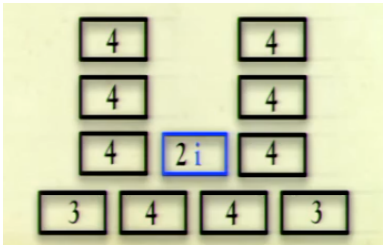
Suppose there is a set $I = \{1, ..., n\}$ of classes that request to use a room. Class i has start time s_i and finish time f_i. We want to book as many classes into the room as possible. However, we cannot book 2 classes that need to use the room at exactly the same time.

What kinda of greedy alg. would we use for this? Here are some options:

1. First-Start: we select the class that starts earliest and iterate on the remaining classes that do not conflict with this selection. This algorithm is bad and doesn't work. For example, the first class could be all day long

2. Shortest duration: we select the class of shortest duration, and iterate on the remaining classes that do not conflict with this selection. This algorithm doesn't work. For example we could have a bad ordering and thus many clashes.

3. Minimum-Conflict: we select the class that conflicts with the fewest number of classes and then iterate. This algorithm doesn't work.



4. Last-Start: we select the class that starts last and iterate.

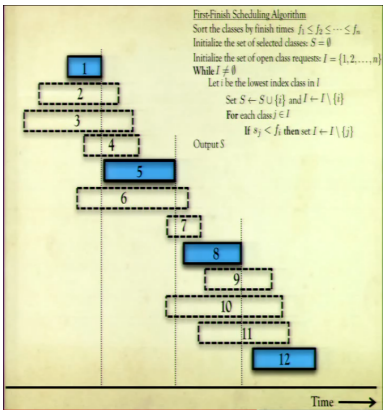
Last-Start actually works and outputs an optimal schedule. It is symmetric to First-Finish where we select the class that finishes first and iterate on the classes that do not conflict with this selection.

Algorithm 1.2.1. FirstFinish(I)

1. Let Class 1 be the class with the earliest finish time.

2. Let X be the set of classes that clash with Class 1.

3. Output $\{1\} \cup$ FirstFinish $(I \setminus X)$.



Theorem 1.2.1. *The FirstFinish scheduling algorithm outputs an optimal schedule.*

Before we prove this, we present the following lemma:

Lemma 1.2.1. There is an optimal schedule that selects Class 1.

Proof. Take an optimal schedule $\mathcal{S}$ with $1 \notin \mathcal{S}$. Let i denote the lowest index class selected in $\mathcal{S}$.

We claim that $(\mathcal{S} \setminus \{i\}) \cup \{i\}$ is a feasible allocation of max. size.

Indeed this follows since: $f_i \leq s_j$ for any $j \in \mathcal{S} \setminus \{i\}$. But $f_1 \leq f_i \leq s_j$ so Class 1 doesn't conflict with any class in $\mathcal{S} \setminus \{i\}$. Thus, $(\mathcal{S} \setminus \{i\}) \cup \{i\}$ is feasible with cardinality equal to $|\mathcal{S}|$. □

Now, back to the proof of the theorem:

Proof. We proceed via induction on $|opt(I)|$. Our base case is that $|opt(I)| = 1$ and clearly FirstFinish outputs $\{1\}$ so we are done. Now our hypothesis is that if $|opt(I)| = k$ then FirstFinish gives an optimal solution. Let $|opt(I)| = k + 1$.

FirstFinish outputs $\{1\} \cup$ FirstFinish$(I \setminus X)$. By our lemma, $\{1\} \in \mathcal{S}^*$ for some optimal solution $\mathcal{S}^*$. Thus $\mathcal{S}^* \setminus \{1\}$ is an optimal solution for the subproblem $I \setminus X$. So $|opt(I \setminus X)| = k$. And by the induction hypothesis, FirstFinish gives an optimal solution for the subproblem $I \setminus X$. $\rightarrow |FirstFinish(I \setminus X)| = k$ and thus

$$|\{1\} \cup FirstFinish(I \setminus X)| = k + 1$$

□

1.2.1 Runtime

Theres are at most n iterations and it takes $O(n)$ time to find the class that finishes earliest in each iteration. Thus, the runtime is $O(n^2)$. But, this was a very crude analysis. With a more subtle implementation and analysis, we can actually get $O(n \log n)$ (exercise!).

2

The Shortest Path Problem

Say that we have a directed graph $G = (V, A)$ with a source vertex $s \in V$. Let each arc $a \in A$ have non-negative length l_a. The length of a path P in the graph is then $l(P) = \sum_{a \in P} l_a$. The problem we consider here is that we want to find the shortest length paths from s to every other vertex $v \in V$. Amazingly, we can actually do this in 1 go, rather than running an algorithm say n times. We do this via the greedy algorithm know as Dijkstra's Shortest Path Algorithm.

2.1 Dijkstra's Shortest Path Algorithm

Set $\mathcal{S} = \emptyset$

Set $d(s) = 0$ and $d(v) = \infty$, $\forall v \in V \setminus \{s\}$

While $\mathcal{S} \neq V$

  If $v = \underset{u \in V \setminus \mathcal{S}}{\operatorname{argmin}} \; d(u)$ then set $\mathcal{S} \leftarrow \mathcal{S} \cup \{v\}$

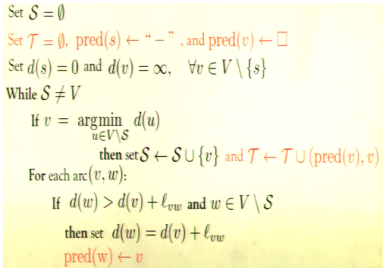
  For each arc (v, w):

    If $d(w) > d(v) + \ell_{vw}$ and $w \in V \setminus \mathcal{S}$

      then set $d(w) = d(v) + \ell_{vw}$

Algorithm 2.1.1.

This algorithm only finds the shortest path distances. But, we can easily enhance the algorithm to keep track of the shortest path themselves as well:



Algorithm 2.1.2.

See the example in the lecture and make sure you understand how this algorithm works!

2.1.1 Special Case reduced to BFS

If each arclength is equal to 1, we simply get BFS. Try it!

2.1.2 Does it Work?

First observe that $\mathcal{S}$ is a set of vertices and $\mathcal{T}$ is a set of arcs. Thus, we introduce the following notation:

1. Let $\mathcal{S}^k$ denote the set of vertices in $\mathcal{S}$ at the end of the k-th iteration.
2. Let $\mathcal{T}^k$ denote the set of arcs in $\mathcal{T}$ at the end of the k-th iteration

Observe that the arcs in $\mathcal{T}^k$ are all between vertices in $\mathcal{S}^k$. Thus, $\mathcal{G}^k = (\mathcal{S}^k, \mathcal{T}^k)$ is actually a directed graph.

Remark. Note that $\mathcal{G} = \mathcal{G}^n$ is the final output of Dijkstra's algorithm.

In fact, it turns out that $\mathcal{G}$ is an **arborescence** (a directed tree) rooted at the source node s:

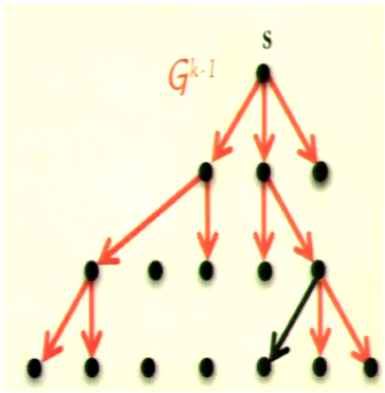
Theorem 2.1.1. *$\mathcal{G}^k$ is an arborescence rooted at s*

Proof. Base case is $k = 1$. Well, $\mathcal{S}^1$ only contains s. But $\mathcal{T}^1$ contains no arcs since $pred(s) =$ "_". Thus, $\mathcal{G}^1$ is trivially an arborescence rooted at s.

Now, our hypothesis is that $\mathcal{G}^{k-1}$ is an arborescence rooted at s. Consider $\mathcal{G}^k$. Let v_k be the vertex added to $\mathcal{S}$ in the k-th iteration. Thus:

$$\mathcal{S}^k = \mathcal{S}^{k-1} \cup \{v_k\} \text{ and } \mathcal{T}^k = \mathcal{T}^{k-1} \cup (pred^{k-1}(v_k), v_k)$$

So, v_k has in degree =1 in $\mathcal{G}^k$ and outdegree = 0. Furthermore, by definition, $pred^{k-1}(v_k) \in \mathcal{S}^{k-1}$. Thus, $\mathcal{G}^k$ is an arborescnece rooted at s with v_k as a leaf.



□

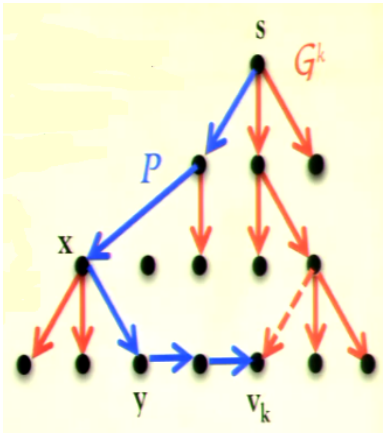
So, at this point, we know that Dijkstra's algorithm outputs an arborescence. But moreover, we claim that the paths induced by the arborescence are shortest paths:

Theorem 2.1.2. *The graph $\mathcal{G}^k$ gives the true shortest path distances from s to every vertex in $\mathcal{S}^k$*

Proof. The base case $k = 1$ is trivial since $d^1(s) = 0 = d^*(s)$. (Here we use the notation that d^* denotes the true shortest path distances.)

Now, for our hypothesis assume $\mathcal{G}^{k-1}$ gives the shortest distances from s to every vertex in $\mathcal{S}^{k-1}$ i.e. $d^{k-1}(v) = d^*(v)$ for all $v \in \mathcal{S}^{k-1}$.

Now, consider $\mathcal{G}^k$. Let v_k be the vertex added to $\mathcal{S}$ in the k-th iteration. We need to show that the new path is the shortest path to v_k. We argue via contradiction. Take the shortest path P from s to v_k with as many arcs in common with $\mathcal{G}^k$ as possible. Let x be the last vertex of $\mathcal{G}^{k-1}$ in P. Let $y \notin \mathcal{S}^k$ be the vertex after x in P. (Such a y $\exists$, or P is in $\mathcal{G}^k$ and we are done).



Since y is on the shortest path from s to v_k and the arclengths are nonnegative, we have:

$$d^*(y) \leq d^*(v_k) < d^k(v_k)$$

noting that we have strict inequality or else we are done. But, since $x \in \mathcal{S}^{k-1}$ we have:

$$\begin{aligned} d^k(y) &\leq d^{k-1}(x) + l(x,y) \\ &= d^*(x) + l(x,y) \quad \text{by the induction hypothesis} \\ &= d^*(y) \end{aligned}$$

Therefore, $d^k(y) \leq d^*(y) < d^k(v_k)$ which contradicts the selection of v_k as $d^k(y) < d^k(v_k)$. □

2.1.3 Runtime

We have n iterations and at most n distance updates at each iteration. Thus, the runtime is $O(n^2)$. Later in this course we will see how to implement Dijkstra's algorithm in $O(m \log n)$ using a heap (here m is the number of arcs).

Acknowledgements

These notes have been transcribed from the lectures given by Prof. Adrian Vetta. Any figures are screenshots of the recorded lectures. This document is for personal use only and beware of typos!